Problem 9 Editorial: SOLIDWORKS is not responding 7+2=9 Points

Problem ID: `extrusion`
Rank: `2+3`

Special thanks to Joshc (Joshua Chen) for contributing to this editorial.

Overview

Problem statement: https://calico.berkeley.edu/files/calico-fa22/contest/extrusion/extrusion.pdf

Although things look a bit scary at first glance, the problem statement straight up gives us an algorithm to draw the extrusion! In fact, a direct implementation of this algorithm is sufficient to solve the main test set. However, the bonus requires observing an optimization that can be made with an insight about overlapping edges.

This insight about overlapping edges alludes to the occlusion problem, a major problem in computer graphics! While determining hidden edges in the extrusion of a simple aligned shape is relatively easy, determining hidden surfaces of complex 3D geometry in unaligned rotations is much more difficult! Modern graphics processors need efficient algorithms to perform operations like these quickly for video games to run at buttery smooth frame rates.

Main Test Set

Cooking by the Book *Solutions available in C++, Java, Python*

The problem statement basically tells us to do the following:

1. Draw the base at the top left of the image
2. For each corner (+ character) in the base, draw an edge with $\mathbf{D}$ backslashes (\ character) towards the bottom right, overwriting characters as needed.
3. Draw the base again at the bottom right, overwriting characters as needed.

We can directly implement this algorithm, storing the image in a 2D array of characters with dimensions $(\mathbf{H} + \mathbf{D} + 1) \times (\mathbf{W} + \mathbf{D} + 1)$.

However, there's a couple things to keep in mind:

- When drawing the edges, it's possible that some corners will be overwritten by previous corners if we're not careful about the order in which we process them. If this happens, we'll miss some edges and hence may forget to still add their respective $\mathbf{D}$ backslashes, thus ending up with an incorrect image. One way to remedy this is to simply make a copy of the base that we read from, separating it from the image we draw on. By doing this, we won't lose track of the corners that get overwritten since we have another copy.
- When drawing the base in the bottom right of our image, we should only draw the corners and edges of the base—don't overwrite existing characters with spaces!

Reading the input and copying the top left base takes $O(\mathbf{HW})$ time. Drawing $\mathbf{D}$ backslashes for the edges of each corner, of which there are no more than $\mathbf{HW}$, takes $O(\mathbf{HWD})$ time. Finally, copying in the bottom right base takes $O(\mathbf{HW})$ time as well. As a result, the complexity for drawing the edges dominates, so we have $O(\mathbf{HWD})$ overall.

Bonus Test Set 1

Since the size of our output is only $O((\mathbf{H} + \mathbf{D})(\mathbf{W} + \mathbf{D}))$, this is also a lower bound for how fast any algorithm can solve this problem. However, our complexity for the main test set of $O(\mathbf{HWD})$ is much bigger than this, and suggests that perhaps we're being inefficient somewhere in the process of drawing the edges. But how? Consider the following example input and output:

```
1
9 9 8
+-+
| |
| +-+
|   |
|   +-+
|     |
|     +-+
|       |
+-------+
```

```
+-+
|\|\
| \-\
|  \|\
|   \-\
|    \|\
|     \-\
|      \|\
+-------\ \
 \       +-+
  \      |\|\
   \     | +-+
    \    |  \|\
     \   |   +-+
      \  |    \|\
       \ |     +-+
        \|      \|
         +-------+
```

If we look at the diagonal from the character in the first row and first column to the character in the last row and last column, we see that there are 5 corners along this diagonal. Critically, this diagonal of the final image has only $\mathbf{H} + \mathbf{D} + 1 = 18$ characters, but since we must draw $\mathbf{D}$ backslashes for each corner, we end up drawing $5 * \mathbf{D} = 40$ backslashes, most of which are redundant overwrites! If we consider the absolute worst case, where every character along this diagonal is a corner , then we would have $\mathbf{H}$ corners, meaning we would need to write $\mathbf{HD}$ characters into $\mathbf{H} + \mathbf{D} + 1$ spaces. Using a similar method of calculation, it's clear that this redundancy creates an asymptotic slowdown and occurs along every diagonal.

With this in mind, we can drastically speed up this process if we have some way of telling where we've drawn backslashes already so we don't waste as much time re-drawing backslashes we already drew.

The Lazy Painter *Solutions available in C++, Java, Python*

One idea is perhaps we can stop drawing backslashes for the remaining edge of a certain corner along some diagonal if we already find a backslash in that position. However, to do this, we need to be careful about the order in which we draw our backslashes. If we go through the corners of the image top to down, left to right, and draw edges towards the bottom right, this fails because we might stop drawing some edges too early.

To remedy this, we can instead go through the corners in reverse, bottom to up, right to left. However, for each corner, we still draw the backslashes of the edges towards the bottom right. Now, if during this process, we are about to overwrite another existing $\backslash$ character, terminate early—even if we have not yet added all $\mathbf{D}$ backslashes. Why? Well, the backslash we were about to overwrite must already start a run of at least $\mathbf{D}$ consecutive backslashes. Thus, we're guaranteed that all $\mathbf{D}$ backslashes for this current corner have already been drawn, so we can safely end the loop early. Then, we perform everything else exactly the same way as the algorithm for the main test set.

Although at first glance, this optimization seems to only provide a constant factor speed-up, it turns out that this lowers our complexity to $O((\mathbf{H} + \mathbf{D})(\mathbf{W} + \mathbf{D}))$, as desired. Consider the entire process of drawing the edges as a whole. Even though we have a nested triple loop, the body of the inner loop only draws a $\backslash$ most once for each character in our image. This is because we would stop early before overwriting an existing $\backslash$. Thus, the runtime for the edge drawing process is proportional to the number of non-backslashes turned into backslashes, bounding the entire time complexity by the number of non-backslash characters, which is at most $O((\mathbf{H} + \mathbf{D})(\mathbf{W} + \mathbf{D}))$.

Reduction to Copypasta *Solutions available in C++, Java, Python*

Another idea is that perhaps we can break the final image down by drawing diagonal by diagonal, instead of row by row or column by column. This is helpful because we can now start by considering all corners along each diagonal, consider the intervals of backslashes each corner will create, merge these relevant intervals together to get rid of overlap, and then finally draw only the merged intervals.

It turns out that merging simple intervals together is a very well known programming problem, and a quick search on the internet will give you a lot of ready-to-copypasta implementations that you can easily integrate into your own code. The only tricky part left of this solution then is to implement diagonal iteration.

What about the complexity of this algorithm? We have $\mathbf{H} + \mathbf{W}$ - 1 diagonals, each with at most $\max(\mathbf{H}, \mathbf{W}) + \mathbf{D} + 1$ characters. The interval merging algorithm given in the article linked above runs in $O(\mathbf{N}\log(\mathbf{N}))$, where $\mathbf{N}$ is the number of intervals, but only because it requires sorting. If we traverse our diagonals in a way such that intervals will be considered in order, we can modify the algorithm to simply be $O(\mathbf{N})$. In either case, the number of intervals we may need to merge, $\mathbf{N}$, is equal to the number of characters along each diagonal, $\max(\mathbf{H}, \mathbf{W}) + \mathbf{D} + 1$, in our case. Thus, we have a final runtime of $\mathrm{O}((\mathbf{H} + \mathbf{W})(\max(\mathbf{H}, \mathbf{W}) + \mathbf{D}))$, which is basically the same thing as our lower bound, and also a significant improvement over $O(\mathbf{HWD})$.